MASTER

Efficiently answering Fréchet queries

van Diggelen, T.W.T.

Award date:
2018

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
Efficiently answering Fréchet queries

Master Thesis

T.W.T van Diggelen

Committee:
Kevin Buchin
Wouter Meulemans
George Fletcher

Supervisors:
Kevin Buchin
Wouter Meulemans

Eindhoven, Monday 18th December, 2017
Abstract

With the recent rise of sensor-sourced positioning data, the need to derive meaningful conclusions from trajectory data increases. A prominent trajectory comparison metric, the Fréchet distance, is often used but is inefficient to compute for large input sets. This thesis attempts to improve the efficiency of Fréchet related computations for representative problems, with representative data. This thesis presents novel efficiency-focused adaptations to existing Fréchet algorithms and their composition into an efficient pruning-based query answering algorithm. The investigated approaches can be integrated piecemeal into other computation systems, and the thesis examines when and how which approach is most effective. The algorithms proposed in this thesis have been shown to be successful in practice, reaching second place in the ACM sigspatial GIScup of 2017.
## Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1</td>
<td>Previous work</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Overview</td>
<td>3</td>
</tr>
<tr>
<td>2.1</td>
<td>Fréchet algorithms</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>Problem set generation</td>
<td>6</td>
</tr>
<tr>
<td>2.2.1</td>
<td>Trajectory data</td>
<td>6</td>
</tr>
<tr>
<td>2.2.2</td>
<td>Pairs problem</td>
<td>9</td>
</tr>
<tr>
<td>2.2.3</td>
<td>Query problem</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>Pruning Methods</td>
<td>11</td>
</tr>
<tr>
<td>3.1</td>
<td>Spatial Hash</td>
<td>11</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Regular grids</td>
<td>12</td>
</tr>
<tr>
<td>3.1.2</td>
<td>Quad tree</td>
<td>13</td>
</tr>
<tr>
<td>3.1.3</td>
<td>Spatial hash experiments</td>
<td>13</td>
</tr>
<tr>
<td>3.2</td>
<td>Simplifications</td>
<td>16</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Greedy simplification</td>
<td>17</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Progressive greedy simplification</td>
<td>17</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Simplification experiments</td>
<td>18</td>
</tr>
<tr>
<td>3.3</td>
<td>Greedy algorithm</td>
<td>27</td>
</tr>
<tr>
<td>3.3.1</td>
<td>ETD</td>
<td>27</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Valley searching</td>
<td>29</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Greedy experiments</td>
<td>29</td>
</tr>
<tr>
<td>4</td>
<td>Fréchet optimizations</td>
<td>31</td>
</tr>
<tr>
<td>4.1</td>
<td>Queue algorithm</td>
<td>31</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Column based Fréchet algorithms</td>
<td>31</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Calculating reachability</td>
<td>32</td>
</tr>
<tr>
<td>4.1.3</td>
<td>Space efficiency</td>
<td>32</td>
</tr>
<tr>
<td>4.1.4</td>
<td>Queue experiments</td>
<td>32</td>
</tr>
<tr>
<td>4.2</td>
<td>Freespace jumps</td>
<td>34</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Computing freespace jumps</td>
<td>35</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Jump graph</td>
<td>37</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Freespace jump experiments</td>
<td>37</td>
</tr>
</tbody>
</table>

Fréchet computations
Chapter 1

Introduction

With the recent increase in positioning enabled sensors, fields like the internet of things and mobile have produced a global ecosystem of trajectory-data-generating devices. Together, the data produced by these devices is comprehensive and indicative of the use and function of these devices themselves directly, as well as their users indirectly. In order to extract interesting conclusions from this data, it is necessary that the trajectory information stored in this data can be processed with high quantity and quality, that is, processed both quickly and with little loss of information. An elementary problem in processing these data is the problem of comparing two or more trajectories efficiently for similarity, with an appropriate metric.

There exist many metrics indicating similarity between polygonal trajectories, but the most prominent one is the Fréchet distance. The Fréchet distance is considered a high quality trajectory comparison metric because it considers the order of vertices in the trajectory, unlike other metrics like the Hausdorff distance which only considers extreme points along the trajectories regardless of order, and does not compute a cumulative error but a maximum error, unlike dynamic time warping which accumulates relative error distances at a loss of information. Finally, the Fréchet distance satisfies the triangle inequality, which is used in optimization strategies throughout this thesis to arrive at conclusions about trajectory comparisons without needing to calculate their (costly) Fréchet distance. Because the Fréchet distance is so costly to compute, the goal of this thesis is to reduce the computation time needed to use the Fréchet distance in practical analysis, regardless of whether we compare the decision problem or the actual Fréchet distance, and regardless of whether we compare one to one, one to many or many to many.

1.1 Previous work

A great deal of work has been put forward to analyse, solve and improve upon the Fréchet distance metric. The introduction of the Fréchet decision problem solution by Alt and Godau [3], using a freespace diagram and corresponding column based algorithms, sparked interest into the Fréchet distance as a computable metric for comparing trajectories. The associated runtime complexity was a problem, however, motivating some work to reduce this in practice with, for instance, pruning measures and approximation algorithms. An obvious motivation for pruning measures appeared in the work of Bringmann et al.[6], who found that if the Strong Exponential Time Hypothesis holds, the Fréchet distance cannot be computed in strongly subquadratic time, which means that pruning algorithms are likely the only way to prevent this quadratic runtime.
complexity in practical problems. Previous work has partially addressed this need for pruning. The process of simplification under the Fréchet distance, as most relevant to this thesis, is posed by Agarwal et al. [3], and later improved for infinite sequences using a streaming algorithm [1].

Other work more tuned toward efficiency include approximations to the Fréchet distance [13] [7] [12], using Fréchet lowerbounds [4], improving the existing Fréchet calculation bounds using a randomized algorithm [8], the postulation of a discrete variant of the problem [14] which can be computed exactly in quadratic time, but has unbounded error when compared to the continuous Fréchet distance, and various datastructure solutions to answer approximate Fréchet queries [10], queries on subtrajectories [11], or queries on geometric trees [15].

Unfortunately none of these cases can be directly applied to the goal of this thesis, and especially research into answering Fréchet query problems exactly but efficiently using an algorithm composed of different optimization strategies is unavailable. Since answering queries is the main goal of this thesis, this lack of research presented itself in prominent ways. Early on, it became apparent that comprehensive and correct datasets, querysets, or even comparison sets do not exist and so must be constructed before the Fréchet query problem itself can be analysed. Moreover, many algorithmic concepts developed around the Fréchet distance, such as simplification, are not constructed with computational efficiency in mind, but instead try to present approximate solutions or solutions focussed on attaining some geometric notion of quality.

Because of this, the following overview section shows how we can first construct meaningful problem sets, so we can later use these problem sets to realistically evaluate pruning methods and optimization strategies.
Chapter 2

Overview

This chapter will give an overview of how the problem of trajectory comparison relates to this thesis, the scope of this thesis, and a more thorough investigation of the Fréchet problem itself. This chapter will give a sufficiently detailed overview of the problem to motivate the more theoretical and experimental choices in the coming chapters.

The Fréchet distance ($\delta_f$) is a trajectory comparison metric developed by French mathematician Maurice René Fréchet, and is generally considered to be a trajectory comparison metric of high quality. We will define the Fréchet distance formally in this chapter as well, but first a more intuitive explanation, see Figure 2.1. Imagine two polygonal trajectories $P$ and $Q$, so two paths consisting of points with straight lines connecting the points. Now imagine a man on path $P$, and a dog on path $Q$. The goal of both the man and the dog is to reach the end of their path. The paths themselves may cross, but the man and dog may never switch paths, nor may they move backwards: they may only move forwards or stand still. Between the man and the dog is a leash. The Fréchet distance then is defined as the minimum length of this leash, such that they may both reach the end of their respective paths. This length is called $\Delta$, and is the solution to the problem.

![Figure 2.1](image)

Figure 2.1: Illustration of the intuition behind the Fréchet problem: a man and his dog, both traversing a path, and the leash (in red) between them. In this explanation the minimum leash length is the Fréchet distance. The man and dog may only go forward or stop, they may never go backwards, and must both reach the end of their own path. Figure (a) shows both the man and the dog in the starting position, in (b) the dog has moved, increasing the leash length.
Given these definitions, the Fréchet problem can be posed in two ways. First, we can ask, as in the example, what the minimum length leash $\Delta$ would be for a given pair of trajectories. Besides this question, we can also ask whether the man and the dog can complete their paths, given a leash of length $\Delta$. This is called the decision problem, and is simpler to compute. Because the problem of minimum leash length can be decomposed in many instances of the decision problem (if bounded error is acceptable), this thesis concerns itself only with the decision problem.

The decision problem can be solved using a so called freespace diagram, see Figure 2.2. A freespace diagram is a two-dimensional matrix which given two trajectories $P$ and $Q$ and a Fréchet distance $\Delta$, contains for each combination of points on $P$ and $Q$ whether $\delta(P, Q) \leq \Delta$, and if so, marks it as free (white). The freespace relates to the Fréchet decision problem as follows: if a monotonically increasing path in $x$ and $y$ can be found through the freespace, such that this path starts in the lower left corner and end in the top right corner, then the Fréchet distance between $P$ and $Q$ is lower than $\Delta$, and vice versa.

![Figure 2.2: An example freespace diagram. If a path is found between the lower left corner and the upper right, through the free space (in white), then the Fréchet distance between the trajectories is lower than the query delta. In this case, the answer to the decision problem should be: true, since such a path exists.](image)

Finally there is the discrete Fréchet distance. When considering the original example of the man and the dog, the discrete Fréchet distance is an adaptation of the example where the man and the dog may only appear at the endpoints of a line segment, and never in between endpoints, on the segment itself. This restriction makes the discrete Fréchet distance easier to compute, which is why it is not the focus of this thesis. Although the discrete Fréchet distance is an upperbound to the continuous Fréchet distance, it is not investigated as an optimization in this thesis because the error of the approximation is unbounded, and therefore unusable.

Now that we have an algorithmic basis for solving Fréchet problems, we can differentiate exactly what these Fréchet problems will look like. First there’s a basic metric comparison: calculate the Fréchet distance between trajectories $P$ and $Q$. Then there’s the problem of finding all trajectories $P$ from a database with Fréchet distance $\Delta$ or lower from query trajectory $Q$, we call this
the *query problem*. Finally, there's the problem of calculating the Fréchet distance between all pairs in a database, we call this the all-pairs problem, answering or approximating this problem is very useful for clustering trajectories, a common problem in computer science. Note the query problem and cluster problem consist of comparisons themselves, which makes them different forms of the comparison problem. Because of this, we will run benchmarks primarily on the querysets.

Given the different practical computational problems arising from instances of the Fréchet problem, this thesis is divided in two major sections: the pruning section, which details how we can reduce the number of Fréchet comparisons that we need to solve when solving any problem except the pair problem, and the optimization section, which details how we can optimize the calculation of the Fréchet decision problem itself, or more specifically, the calculation of the freespace diagram, for the pairs which cannot be pruned.

### 2.1 Fréchet algorithms

As we are concerned with improving the efficiency of Fréchet calculations, we must first establish a base of Fréchet algorithms which effectively solve the continuous versions of the problem. The continuous Fréchet decision problem is solved by solving the freespace diagram with a column algorithm by Alt and Godau [3], the algorithm works by computing all free and reachable space in a freespace diagram, by processing the cells from bottom left to top right. We will call this process a column algorithm in the future. Approximating the Fréchet distance can now be done by binary searching the $\Delta$ space using this algorithm, stopping the search when we have reached the desired accuracy. We can also solve the Fréchet distance problem exactly, but this is slow, and the results of any such algorithm will still be bound in error to the precision used by the computer executing the algorithm, which makes the process less interesting in practice.
2.2 Problem set generation

In order to test the effectiveness of the various approaches and optimizations for solving Fréchet problems, we have to obtain verifiable input and output that are representative of real world data, incorporate edge cases, and be comprehensive and large enough to test performance against. As mentioned, such sets cannot be easily obtained from previous work. Generating problem sets that meet these criteria is the first step to obtaining reliable experimental performance statistics. This section will describe how these problem sets were produced, and what properties they have. Because all results and conclusions in this thesis will only be valid for the problem sets, and not necessarily any other trajectory data, it is paramount that these problems are representative of real world trajectory problems. We derive these problem sets from databases of trajectories, each database having different properties.

2.2.1 Trajectory data

The three Fréchet problems all need a common set of input: trajectory data. For this thesis, three trajectory data sources are used, each having different properties. This section will explain the differences between the datasets. The three datasets are:


The term *complexity* is used to indicate how much a trajectory differs from a straight line, a complex trajectory is very different from a straight line, and may contain (for instance) loops. This metric is defined more formally in Section 2.2.1, but the intention of the term is to capture an intuitive notion of how the shape of a trajectory may influence the difficulty of analyzing it. As such, the term has no mathematical basis or justification.

The term *dimension* is used to indicate the size of the database in the coordinate set it is provided in. A dataset covering a large area would have large dimension. Technically, since any dataset can be scaled to the same dimension, this metric should not influence the result of any analysis. The use of different dimensions for the datasets is a validation that the analysis used is indeed dimension-agnostic.

These datasets were all used in the 2017 Sigspatial Cup as challenge data sets. These databases are different enough in scale and shape that they should represent real world data, and are in fact themselves derived from real world data. To normalize the problem sets across the different databases (which vary in number of trajectories), we normalize the size of the Databases to 10,000, Databases which have insufficient trajectories receive duplicate entries. This does not affect the quality adversely, as the algorithms do not exploit any loopholes that arise from duplicates, nor suffer any of the side effects. Note that this feature affects the results somewhat, as

---

1[^sample]: [http://www.martinwerner.de/files/dataset-sample.tgz](http://www.martinwerner.de/files/dataset-sample.tgz)
for instance many query trajectories on the character database will return a trajectory even with \( \Delta = 0 \), because of these duplicates. This normalization is done because it levels the playing field for comparing algorithms that depend on number of trajectories for their runtime complexity, which can be expected. Also, it makes it easier to mentally compare the difficulty the computer has with solving a problem for a database, given some run time in seconds.

Although the dimensions of the dataset should not influence the problem, it is important to verify that the solutions are dimension-agnostic in practice. This is not necessarily true by default, as certain choices of algorithm parameters may influence performance based on dimension, and computer precision issues may arise.

**Trajectory length**

Figure 2.3 shows the trajectory length distribution, as a histogram, for the three trajectory databases. The trajectory length is measured as the number of vertices in the trajectory.

Fréchet computations
CHAPTER 2. OVERVIEW

Figure 2.3: Trajectory length histogram for each trajectory database. For example, in the sample dataset (a), around 3% of all trajectories (all test cases) are of length 100. Note that the databases not only have different average length, but also different distributions, especially the tdrive database cannot be accurately described using an average length, as the spread of the distribution is huge.

Trajectory complexity

As a simple comparative measure of trajectory complexity we can use the ratio of trajectory length vs the diagonal of the axis aligned bounding box of the trajectory, called $c$. Contrary to the previous section, trajectory length means the geometric length of the trajectory: the sum of all segment lengths included in the trajectory. By this metric, a minimum-complexity trajectory $T$, a trajectory which is simply a line, would have $c = \frac{|T|}{bbox(T) \cdot \text{diagonal}} = 1$. More complex trajectories have a higher value of $c$. Note that a high $c$ does not necessarily mean that this trajectory is complex from a Fréchet standpoint, that it is difficult to do Fréchet comparisons or computations with. What can be concluded, is that the shape of a trajectory with high $c$ is likely to be more difficult to effectively simplify, which may make it more difficult to prune. Figure 2.4 shows histograms of the trajectory complexity for the different databases.
2.2. PROBLEM SET GENERATION

(a) Sample dataset

(b) Tdrive dataset

(c) Wchars dataset

Figure 2.4: Complexity histogram for each dataset. For example, in the sample dataset (a), around 2.5% of all trajectories (all test cases) have a complexity rating of 1.5. Note the differences in scale on the horizontal axis, which shows that the tdrive database is many times more complex (according to the proposed metric) than the other databases.

Because of the different Fréchet problems described before, we compose two different types of problem sets from these databases.

2.2.2 Pairs problem

The most elementary Fréchet problem, whether a decision or actual distance computation, is the pairwise problem. The pairwise problem consists of a problem tuple (or pair) \((P, Q, \Delta)\), where \(P\) and \(Q\) are trajectories and \(\Delta = \delta_f(P, Q)\). These pairs can be used to verify algorithm output as well as run performance statistics. It’s important that these pairs are generated such that they are not all trivial, or all very difficult to solve, but have a range of difficulty. Furthermore, the pairs should all be sufficiently different.

Because the input trajectory databases are sufficiently varied, pairs selected uniformly at random from these databases are also sufficiently varied. What remains is to choose an applicable

Fréchet computations
problem set size, which is mostly benchmarking related. We chose to use 10000 pair problems for each set. As such we end up with three sets of pair problems, one for each trajectory database. For each pair problem, the Fréchet distance is precomputed for verification purposes and so the problem set is a table of \((P, Q, \Delta)\) tuples.

### 2.2.3 Query problem

The query problem consists of a set of tuples \((Q, \Delta)\), with \(Q\) a trajectory, and \(\Delta\), a query Fréchet distance threshold, that serve as queries to be run on a database \(P\), where the queries themselves are to be representative, and the database of of sufficient size to motivate using optimization data-structures to answer the queries. Each query \((Q, \Delta)\), when fed to the algorithm, should produce a list of output trajectories \(L\) with \(L \subseteq P\) were for each \(p \in L\), \(\delta_f(p, Q) \leq \Delta\) and for each \(s \notin L, s \in P\), \(\delta_f(s, Q) > \Delta\).

Generating problem sets for the query problem is more complicated than the previous problem, because the complexity of the query problem set is not only determined by the complexity and size of the trajectory database, but also by the combination of the query trajectory and the query delta.

A trivial example of this is a query with \(\Delta = 0\), which is efficiently answered regardless of which trajectory is the query trajectory, or what is contained within the database. In a more complicated example, if a query trajectory contains no other trajectories within Fréchet range \(\Delta\), queries with that \(\Delta\) are likely to be easy to answer. Moreover, if many trajectories \(T\) in database \(P\) have a Fréchet distance close to \(\Delta\) with a query trajectory \(Q\), answering the query may be complicated because many approximation algorithms may not be able to decide, because the small difference in Fréchet distance is occluded by the introduced error. In conclusion, selecting good queries requires investigation of the dataset, this subsection will describe how we do this investigation, and what the resulting problem set looks like.

For generating query problem sets, we use the following assumption: if a database \(P\) is varied enough, and a query set is large enough, and the number of trajectories returned by the queries are varied enough, it is a good query problem set. Given that we already have varied databases, and we can simply select a large enough query problem set, we only need to ensure that for each query the chosen epsilon returns a varied number of trajectories, and the problem set will be good enough.

We construct the problem set for \(P\) as a list of tuples \((Q, \Delta, n)\) with \(Q\) as the query trajectory, \(\Delta\) as the query delta and \(n\) as the (approximated) number of trajectories the query returns. We then construct tuples such that the problem set has a suitable distribution of returned trajectories, and we construct tuples by binary searching the epsilon space for each chosen query trajectory. The query trajectories are chosen at random from the input database. More formally, we achieve this by picking, for each query trajectory \(Q\), a return percentage \(p\). Now, to turn \((Q, p)\) into \((Q, \Delta)\), we need to pick \(\Delta\) such that \((Q, \Delta)\) returns \(p \times |P|\) trajectories. This can be achieved by binary searching the query delta space given the query trajectory, running a query algorithm and finding the number of results. Note that we vary this ratio based on the complexity of the dataset, because a query returning 5% of the database with simple trajectories may be very common, while it may be very rare in more complex datasets. The ratios chosen are:

- Sample: 0.01% Tdrive: 0.002% Wchars: 0.01%
Chapter 3

Pruning Methods

Now that we’ve established the problem, we can look at the various approaches we can use to speed up any Fréchet distance computation. In doing so, we make a distinction between pruning the input problems and optimizing the Fréchet distance freespace calculation. The act of pruning is the attempt to avoid computing the Fréchet decision problem by using approximation algorithms and see if they succeed. Different pruning approaches can have varying degrees of effectiveness and efficiency, but they should all be more efficient than the Fréchet decision computation. This chapter contains all pruning approaches considered for this thesis, and investigates their effectiveness through experiments.

3.1 Spatial Hash

When calculating the Fréchet distance between trajectories \(P\) and \(Q\), the distance between the endpoints of \(P\) and \(Q\) can be used to derive a lowerbound for the problem. To revisit the original analogy, the man and the dog must both start at the starting points, and end at the endpoints of their trajectories in order to complete their walk, regardless of how they walk. These endpoint distances are then the minimum size of the leash needed to traverse the trajectories, and so are a lowerbound for the Fréchet distance. More formally, this lowerbound can be computed as follows. Consider the first points \(P_{\text{start}}\) and \(Q_{\text{start}}\), and the last points \(P_{\text{end}}\) and \(Q_{\text{end}}\), we can derive a lowerbound using the formula \(\delta_f(P_1, Q_1) \geq \max(\delta(P_{\text{start}}, Q_{\text{start}}), \delta(P_{\text{end}}, Q_{\text{end}}))\), where \(\delta\) denotes the Euclidean distance and \(\delta_f\) the Fréchet distance. This means that for any Fréchet decision problem instance \((P, Q, \Delta)\), the problem can be answered as \textit{false} in constant time if the query delta is smaller than the previously mentioned lowerbound.

More generally, given a database \(\mathcal{P}\) and a query trajectory \(Q\) and query delta \(\Delta\), we can discard all trajectories from \(\mathcal{P}\) which do not meet this lowerbound. This can be answered quickly using two range queries of size \(\Delta\) on the endpoints of the query trajectories, any trajectory which is not a result of both range queries does not meet the lowerbound. To avoid doing the constant time lowerbound computation on all trajectories in \(\mathcal{P}\), we can use a spatial hash. Two candidate spatial hashes are regular grids and quadtrees, which will be explained in the following sections.

Similarly to the lowerbound derived from the endpoints, we can construct a lowerbound from the trajectory bounding box (axis aligned)[5]. Given two trajectories \(P\) and \(Q\), we know that the Fréchet distance between \(P\) and \(Q\) must be at least the distance between the top borders of the bounding boxes, because that is the minimum leash length that would need to occur at least once.
to traverse both trajectories. Similarly, the left, right and bottom sides of the bounding box can be used to the same effect. This lowerbound can be used for pruning trajectory comparisons, just like the previously mentioned endpoints. We will refer to this type of pruning as “Boundingbox pruning”.

3.1.1 Regular grids

The simplest type of spatial hash is a regular grid, see Figure 3.1. In a regular grid the two dimensional plane is divided into equally sized cells, where each cell contains a list of all entities that fall within the boundaries of that cell. In the Fréchet range queries problem, all trajectory endpoints are inserted into their corresponding cells in the grid as a preprocessing step. Then, for each query \( (Q, \Delta) \), we consider only the cells within Euclidean distance \( \Delta \) from \( Q_{\text{start}} \) and \( Q_{\text{end}} \). We can now intersect the trajectories of the two groups of cells to find all trajectories with \( \max(\delta(P_{\text{start}}, Q_{\text{start}}), \delta(P_{\text{end}}, Q_{\text{end}}) \leq \Delta \). All other cells, and the corresponding trajectories, can be ignored.

Note that the trajectories resulting from these range queries only meet the lowerbound for the Fréchet distance, they do not necessarily meet the query threshold \( \Delta \). Regular grids are efficient because determining the range of interesting cells is a constant time operation. This comes at the cost of having uniform granularity: if a large part of the two dimensional plane is unused, the regular grid will allocate just as many cells to it as a more densely populated part of the plane. This wastes memory, but also computation power.

To understand the limitations of a regular grid, imagine the case where a query range is much smaller than the size of a cell, but falls on the corners of four highly populated cells. In this case, the query must compute the euclidean distance for each point in the four cells, only to find a very small subset of the cells which actually satisfy this constraint. This defeats the goal of having a spatial hash, since the spatial hash is used to discard many items quickly without manually inspecting them one by one. Hence, it is important to select a proper grid size.

Unfortunately this is not always possible. In a situation with many outliers, the dimensions of the space that the regular grid must span may be large in comparison to the division of points over that space. In this scenario, many cells have little or no points inside, while very few cells have many points. Because the regular grid scales quadratically with the dimensions of the cells, simply choosing a smaller cell size in these cases results in greater memory use. These issues are not easy to fix using a regular grid, and so the constant time lookup comes at a price.
3.1. SPATIAL HASH

(a)

Figure 3.1: Using a regular grid for pruning. Shown in gray: a query trajectory, in orange: its endpoints and the range queries around them, in blue: the grid cells it touches.

3.1.2 Quad tree

To solve some of the issues that apply to the regular grid, we can use a quad tree. The quadtree is a recursive version of the regular grid, where cells are divided into four subcells if they receive more than a specified capacity of points. The quad tree handles exactly the cases where the regular grid comes up short: by allocating resources only where more resources are required, the quad tree strikes a balance between using memory and having appropriate cell size. This reduces memory and computation cost, at the price of making insertion and range queries slightly more complicated and costly. The results obtained with spatial hashes in the experimental phase of this thesis were neither beneficial enough nor costly enough to justify investigating more complex spatial hashes like a quadtree, and so this thesis does not explore quadtrees with in depth experiments.

3.1.3 Spatial hash experiments

Because the spatial hash is a pruning method derived from a lowerbound, its performance cannot be measured in terms of effectiveness, only efficiency, since each trajectory comparison either meets the lowerbound or doesn’t, which makes the spatial hash an exact pruning measure, and one hash cannot outperform another in terms of comparisons pruned. In these experiments, we show the pruning percentages and runtime performance of the spatial hash.

Regular grid pruning percentages per dataset

As a base metric of performance for the spatial hash we can look at the number of trajectory comparisons it can discard. This experiment measures the percentage of trajectory comparisons pruned by the spatial hash, in other words, the fraction of comparisons which can be discarded based on the endpoint lowerbound for the Fréchet distance. This experiment is a motivation for the use of a spatial hash, not an evaluation of the spatial hash itself, as the properties shown are derived from the shape of the input data, and any pruning method using these lowerbounds would report identical results.

Fréchet computations 13
CHAPTER 3. PRUNING METHODS

The pruning percentages found, for each dataset:

Sample: 95%

Tdrive: 85%

Wchars: 85%

Bounding box pruning percentages per dataset

The pruning percentages found, for each dataset:

Sample: 95%

Tdrive: 95%

Wchars: 98%

Regular grid cells size

The regular grid is a simple spatial hash, and we need only specify one parameter for it. This parameter is the number of grid cells in the two dimensions: the resolution of the regular grid. Plotted below in Figure 3.2 is, for each dataset, the number of grid cells in one dimension (horizontally) and the experiment time in milliseconds (vertically). The experiment itself consists of running all queries in a set, and for each query determining the number of trajectories that meet the lower bound.

The results for the sample dataset show a clear trend where increasing the number of cells reduces the runtime of the experiment, up to some point. This can be expected, as reducing the cell size reduces the number of constant-time computations that the regular grid needs to perform, as more and more trajectories fall in cells outside of the query range. Increasing the cell size too far, conversely, means that the regular grid spends more time figuring out which cells are relevant and accessing them, while the resulting number of useless constant computations reduces only minimally.

Unfortunately the other two datasets do not exhibit this behavior; this can be explained by the relatively large query delta size compared to the dimensions of the dataset. This has the result that, regardless of the resolution of the regular grid, nearly all trajectories must be considered in constant time.

14 Fréchet computations
Figure 3.2: Effect of varying regular grid cell size on total pruning time performance, for each dataset. For example in the sample queryset, a grid dimension of 30 cells results in a pruning time of 300ms.
CHAPTER 3. PRUNING METHODS

3.2 Simplifications

A trajectory in black, and a simplification in red. Notice how the red trajectory follows the shape of the black trajectory, but uses less vertices.

Figure 3.3: A trajectory in black, and a simplification in red. Notice how the red trajectory follows the shape of the black trajectory, but uses less vertices.

A different kind of pruning technique is the use of simplifications, see Figure 3.3. Simplification is the process of reducing the number of vertices in a trajectory while retaining the overall shape of the trajectory. It is an attempt to construct a new simpler version of the original trajectory, which looks like the original trajectory. We can use the result of Fréchet computations on these simplifications to say something about the original trajectory. The advantage of using simplifications is then that the computations done on the simplifications are faster because of their reduced size. Specifically, solving the freespace diagram is a quadratic operation, so reducing the input size can be very helpful.

The way we draw conclusions about the original trajectory, given the simplification, is via the triangle inequality. If \( P \) and \( Q \) are original trajectories, and \( P' \) and \( Q' \) are their simplifications, we can use the triangle inequality as follows. First we determine the simplification errors \( \varepsilon_p = \delta_f(P, P') \) and \( \varepsilon_q = \delta_f(Q, Q') \), these numbers are a metric of the quality of simplification. We can then construct the implications

\[
\delta_f(P', Q') + \varepsilon_p + \varepsilon_q \leq \Delta \Rightarrow \delta_f(P, Q) \leq \Delta
\]

and

\[
\delta_f(P', Q') - \varepsilon_p - \varepsilon_q > \Delta \Rightarrow \delta_f(P, Q) > \Delta
\]

In natural language, these equations say that the Fréchet distance between the simplifications is indicative of the Fréchet distance between the original trajectories, taking into account an error bound of \((\varepsilon_p + \varepsilon_q)\). So unfortunately the simplification process introduces an error to the decision problem, which can leave the decision problem undecided. This happens when trajectories are very similar in relation to the decision \( \Delta \). When this happens, the computation has not yielded a usable result, and we need to try something else. In practice this means either using a more detailed simplification (with a corresponding smaller error bound), or another pruning step altogether. The coming sections will explain in detail the simplifications used for this thesis.
3.2. SIMPLIFICATIONS

3.2.1 Greedy simplification

In order for us to use the triangle inequality as a method of answering the decision Fréchet problem, we need to know the Fréchet distance of the simplification to the original trajectory, called the simplification error in the introduction of this section. Given any simplification $P'$ of $P$, we can construct this error by calculating $\delta_f(P, P')$ at runtime cost of $O(|P| \ast |P'|)$, but this is unnecessary. Alternatively, we can use a simplification algorithm which may produce an error bound for us, specifically the simplification algorithm by Agarwal et al. \[2\]. In this algorithm, a specific error threshold $\varepsilon$ is supplied to the algorithm and the algorithm ensures that the simplification error does not exceed the threshold. In other words, the algorithm produces a simplification with the guarantee that $\delta_f(P, P') \leq \varepsilon$. The additional computational cost of calculating the Fréchet distance between $P$ and $P'$ can now be avoided, and the supplied $\varepsilon$ can be used in the triangle inequality. Even though $\varepsilon$ is not the Fréchet distance, it is an upperbound to the Fréchet distance, so it can still be used in the triangle inequality.

In order to create a simplification that meets this bound, the simplification algorithm by Agarwal et al. uses a decision Fréchet computation internally to constantly check that it does not exceed the bound. This is once again a quadratic operation, which is undesireable. Our adapted simplification algorithm omits this quadratic operation by using a linear time greedy approximation algorithm called Equal Time Distance instead of the decision algorithm, which calculates an upperbound to the Fréchet distance. Because this is an upperbound, the increased efficiency of the linear algorithm comes at the cost of increasing the simplification error, but this difference is negligible, see Figure 3.11.

3.2.2 Progressive greedy simplification

Even though using the Agarwal et al. algorithm has many benefits, it requires us to supply a simplification threshold $\varepsilon$. Deciding on a good threshold can be difficult, because the reason we use simplification is not necessarily to produce a good looking but simple trajectory, but to construct a simplification which can help us answer the original problem more efficiently. This means that we don’t really care about $\varepsilon$, we care about the efficiency of the resulting trajectories when we do Fréchet computations with them. Since the Fréchet computation scales with input trajectory length, we need to produce simplifications of a particular length to achieve a particular efficiency. Unfortunately, the simplification algorithm itself does not give us a bound on simplification length, and there is no direct correlation between $\varepsilon$ and length, as it can vary based on trajectory shape, dimensions and length. Luckily, $\varepsilon$ and length are correlated in the sense that a simplification constructed with a larger $\varepsilon$ will always have fewer vertices. In other words, the lower the number of vertices in a simplification produced by the algorithm, the larger the error in the triangle inequality, and the larger the chance that the decision problem remains inconclusive. This means that if we set a target trajectory length, we achieve a direct correlation between computational efficiency and introduced error (which means pruning or decision effectiveness).

To solve the issue of matching desired length to a simplification threshold, we can do a binary search the $\varepsilon$ space and track the size of the trajectory that comes out. We start with the original trajectory $T$ and a desired simplification ratio $r$, we then construct simplification $T'$ such that $|T'|/|T|$ approaches $r$. Since we will never get exactly $r$, we can experimentally determine how many binary search steps we want to invest, or indicate a certain precision of $r$ that we wish to approach. The $\varepsilon$ space spans from 0 to $\infty$ for a random trajectory, which is too large to search, so we can instead try to determine minimum and maximum values limiting the $\varepsilon$ space. In our

Fréchet computations 17
approach, the minimum and maximum used are based on the diagonal of the bounding box of the trajectory, which is effective in practice. Nevertheless this operation is relatively inefficient and inelegant, and could be improved by a more clever construction method.

Now that we have a strategy which can produce a simplification with a given target trajectory size, we need to pick a useful simplification ratio $r$. Picking a single such value is complicated, because the complexity of a given trajectory comparison might make it more beneficial in some cases to spend more time to achieve a lower error bound, while for other cases this error bound may be much too small, causing us to lose efficiency. Instead, we can choose to construct a number of progressive simplifications, ordered from coarsest to most detailed. This way, we first do a simple computation, and only do more complicated computations if the simple computation is inconclusive. This raises the question of exactly how many simplifications, and what ratios they should have, these questions are answered in the experiments section.

3.2.3 Simplification experiments

In these experiments, we will look at the effectiveness of simplifications, specifically determining the most effective simplification ratios, and we analyse the effect of using a greedy Fréchet proxy in the simplification algorithm.

**What simplification percentage per simplification level**

As explained previously, the efficiency of the algorithm requires us to determine a specific ratio of simplification. That is, the size difference between the simplification and the original trajectory, as a ratio.

The experiment used in this subsection works as follows. For a given query set, we first construct a set of simplifications for the corresponding trajectory dataset. Then, for each query, we apply the same simplification to the query trajectory, and attempt to prune using these simplifications. The obtained pruning percentage, as well as running time, is used to analyse the results. For all tests, it is ensured that the comparisons are at least interesting, meaning the query delta is greater than the largest endpoint distances of the comparison (otherwise the query is answerable in constant time, regardless of used simplification).

First, we can observe that the simplification does indeed become more successful in pruning as it approaches the size of the original trajectory. Figure 3.4(a) shows clearly that increasing trajectory size increases the pruning effectiveness, with 80% of interesting comparisons decided using only 8% of the original trajectory length. The pruning metric itself is not sufficient though, because increasing trajectory length also means increasing the cost of the Fréchet decision computation (figure 3.4b), as well as the cost in constructing the simplification during preprocessing (figure 3.4c).
3.2. SIMPLIFICATIONS

Choosing a good simplification level based on these graphs is difficult, the choice seems to be arbitrary: a higher simplification ratio results in more trajectories pruned, but also takes more time. In order to make an informed decision, we must measure the value of the trajectories which are pruned, or better yet: the trajectories which are not pruned. The reasoning behind this is that the trajectories that cannot be pruned by the simplification algorithm will need to be examined with, and decided by, a more complicated and more costly algorithm. We can solve the issue by introducing a plausible cost function for the trajectories which are not pruned by the simplification step. This cost function should be realistic, such that it does not overestimate or underestimate the real cost of comparing the trajectories, and so is able to estimate the difficulty of solving a trajectory comparison. We could attempt to construct a metric for this, or create a complicated learning process to solve the problem exactly, but it is far simpler to use the Fréchet decision problem calculation itself. More specifically, we use as a cost function exactly the algorithm that would run in our complete problem solution if a comparison would fail.
CHAPTER 3. PRUNING METHODS

We can now analyse the behavior of the resulting algorithm, this algorithm consisting of a simplification pruning step and a Fréchet decision problem step. The result of this analysis is shown in Figure 3.5. At first sight, this graph may not seem to be much more interesting than the previous graphs, because it does not show a clear minimum. However, compared to the previous graphs this new analysis shows us a clear point where further increasing the simplification ratio does not result in an improvement in computation time (3.5c). This is a result of introducing the cost function: it shows us the point where the increased pruning percentage, in combination with the increased computation time, does not outweigh the savings in the cost function for the additionally pruned comparisons. For completeness, 3.5 shows the preprocessing time (a), pruning time (b) and total time (c) of the experiment.

![Figure 3.5: The behavior of the simplification step with a cost function, measured against a simplification ratio between 0 and 80% of the source trajectory, on the sample queryset.](image)

From figure 3.5c we can conclude that the simplification ratio of the first level of simplification should be 25%, but this is not directly true, there may be a more efficient simplification ratio smaller than 25%, which, in a simplification pruning procedure, would have to be executed before the simplification with ratio 25%. We can arrive at this conclusion by executing the following Fréchet computations.
3.2. SIMPLIFICATIONS

experiment: given the knowledge that 25% is a good simplification ratio (for the sample queryset), we can attempt to construct another simplification level, with a ratio somewhere between 0 and 80%, and see how the complete algorithm performs. The result of such an experiment can be seen in Figure 3.6.

Figure 3.6: Obtaining an optimum simplification ratio for a new level, given the previously determined optimal simplification ratio of 25%. For this experiment, we execute three steps: first a 25% simplification pruning step, then a simplification pruning step with $x\%$ ($x$ plotted on the x axis), and finally the cost function.

Figure 3.6 shows us that, given an existing simplification level of 25%, if we desire another level of simplification it should in fact be below 25%, hence the conclusion that the first level found is not necessarily the first level in the final pruning configuration. To summarize, the results should be interpreted as follows: if only one simplification step is desired, the step should have a ratio of 25%, if two steps are desired, the ratios should be first 10% and then 25%. We can expand this reasoning to obtain a sequence of graphs for the consecutive simplification levels, which can be seen in Figures 3.7 (sample queryset), 3.8 (wchars queryset) and 3.9 (tdrive queryset).

Note that the results are not as clear cut for the tdrive and wchars querysets compared to the sample queryset, other values may be chosen with a better efficiency result. The goal of this process is not to find the optimal values, necessarily, as the approach is wrong for this: the used cost function is simply incorrect, as the cost of failing any simplification level is not necessarily having to calculate the Fréchet distance, but the cost may be simply another pruning step, or another simplification level. Instead, the chicken-and-egg problem resulting from having to construct an algorithm based on these values, and needing to have a constructed algorithm to derive these values, makes this approach the most feasible.
Figure 3.7: Plots of the iterative simplification level analysis for the sample queryset, all plots are of the total time.
3.2. SIMPLIFICATIONS

Figure 3.8: Plots of the iterative simplification level analysis for the wchars queryset, all plots are of the total time.
Figure 3.9: Plots of the iterative simplification level analysis for the tdrive queryset, all plots are of the total time. Note that the problem size has been reduced to 128 queries for these plots.

Using greedy algorithm as Fréchet proxy during simplification

The key improvement of the greedy simplification algorithm is the use of a suitable proxy for the Fréchet distance, which is faster but does not affect the quality of the resulting simplification (too much). To motivate this claim, the experiments listed below show the difference in simplific-
3.2. SIMPLIFICATIONS

Apart from the quality produced by the greedy and non-greedy algorithms, as well as the resulting runtime improvement.

These experiments analyse the simplification of all trajectories in a dataset, using a fraction of the bounding box as a simplification epsilon.

The first set of tests, in Figure 3.10, show a cumulative plot of the number of vertices in the simplification, as a fraction of the number of vertices in the original trajectory. To retain a good quality of simplification, the number of vertices must not differ much by introducing the Fréchet proxy, and especially should not be much bigger, because this negatively affects the runtime of any Fréchet computation done with the simplification. As can be seen from the figures, the resulting number of vertices differs only slightly, with the majority of the trajectories being nearly equal in length.

![Cumulative plot of the ratio of vertices of the greedy simplification.](image)

**Figure 3.10:** Cumulative plot of the ratio of vertices of the greedy simplification. For example, 90% of the trajectories in the tdrive database can be simplified using the greedy algorithm at a maximum of 20% additional vertices compared to the normal simplification. The graphs show clearly that the majority of greedy simplifications are only slightly larger than the traditional simplification.
The second set of tests, in Figure 3.11, show a cumulative plot of the resulting Fréchet distance to the original trajectory as a percentage of the simplification threshold specified. In other words, a simplification $T'$ of $T$, simplified with threshold $\varepsilon$, with $\delta_{\varepsilon}(T, T') = 0.5 \ast \varepsilon$ would, in the histogram, fall in bar 0.5. A good simplification should then be close to 1, since the goal of the simplification algorithm before we injected the proxy, was to produce a simplification which has only as many vertices as the simplification threshold requires. Because we have established that the number of vertices generated is close to the traditional algorithm, the analysis in Figure 3.11 validates the claim that the greedy simplification algorithm produces simplification of sufficient quality, as the Fréchet distance between the simplification and the original trajectory closely matches the specified simplification threshold.

Figure 3.11: Cumulative plot of the ratio of the Fréchet between greedy simplification and original trajectory, and the given simplification threshold. For example in the sample database, half of the trajectories can be simplified with a resulting Fréchet distance 98% of the simplification threshold. The graphs show clearly that the majority of greedy simplifications are very close to the simplification threshold, and so are of high quality.
Now that we’ve shown the greedy simplifications are of comparable quality to other simplifications, we can demonstrate that the use of a proxy indeed improves the simplification time, as expected when migrating from a quadratic to a linear time proxy algorithm. Because of the way the original algorithm uses the quadratic time Fréchet decision algorithm, the speedup itself is not quadratic but rather linear. However, the speedup is still significant, as can be seen in Figure 3.12(a,b,c), with about a factor 2 improvement in running time by using ETD as a Fréchet proxy, regardless of the complexity of the input trajectories or the used simplification threshold.

![Sample comparisonset](image1)
![Tdrive comparisonset](image2)
![Wchars comparisonset](image3)

Figure 3.12: Plotted the relative improvement in time on the $y$ axis when using a greedy algorithm as a proxy for the Fréchet distance when computing simplifications, and various scales of simplification threshold on the $x$ axis. It can be clearly observed that the time improvements hold for most usable simplification epsilons. The simplification thesholds are taken as a fraction of the trajectory diagonal, so a sample of 90 means that the simplification theshold is $\frac{1}{90}$th of the bounding box diagonal.

### 3.3 Greedy algorithm

As previously explained, the Fréchet decision problem can be reduced to solving the freespace of the $(P, Q, \Delta)$ combination. Solving the entire freespace diagram is computationally intensive, which is why the previous sections have tried to avoid this. Where the previous sections tried to reduce the size of the problem to speed up decision making, this section will try to change the problem itself.

The approach to simplifying the decision problem is to attempt a solution which may fail, attempting to find a path through the freespace diagram in an uninformed and incomplete way, on the off-chance that we are successful. Such an approach, relying on an approximated view of the problem using a greedy choice, can be much faster than the full freespace computation because it does not have to produce a conclusive answer.

If it succeeds in reaching an answer, however, it will do so much faster than the quadratic time needed for the Fréchet decision problem. The goal of the greedy algorithm is to be fast but inaccurate, as will be further investigated in the experiments section. This section will look at various greedy algorithms.

#### 3.3.1 ETD

A very simple version of a greedy algorithm is called Equal Time Distance, see Figure 3.13. Conceptually, this algorithm attempts to traverse the diagonal of the freespace diagram. It can either compute the required Fréchet distance to traverse the diagram, or simply answer the decision
problem whether threshold $\Delta$ produces a free diagonal. Because the algorithm only traverses the diagonal, it has a predictable $\Theta(n)$ order running time. Much faster than the quadratic decision problem time.

Equal time distance simply intersects the diagonal of the freespace diagram with every cell, producing a single intersection per cell. This intersection corresponds to a point $p$ on $P$ and $q$ on $Q$. ETD can then check whether $\delta_f(p,q) \geq \Delta$, and return false if this is the case (since the diagonal, at point $(p,q)$, is then not in the freespace. If $(p,q)$ is the end for both trajectories, then the algorithm was successful and can return true.

Figure 3.13: Equal time distance, explained using the freespace diagram. ETD tries to traverse the diagonal, as seen in (a), if it fails to do so (b), we cannot draw any conclusion, as there may be alternative paths (c).

Moreover, Equal Time Distance can be successfully used as a proxy to the Fréchet distance, which makes it even more valuable. We can change the algorithm to not just simplify the decision problem, but also the Fréchet distance computation. We can, when traversing the diagonal, not just stop when encountering a vertex $(p,q)$ where $\delta_f(p,q) \geq \Delta$, but we can choose to always traverse the entire diagonal and remember the maximum encountered $\delta_f(p,q)$. This distance forms an upperbound to the Fréchet distance itself, because we know that it is indeed possible to traverse the freespace diagram given $\Delta$, though there may be a more efficient route (and a corresponding smaller $\Delta$). In formula, we can say that $\delta_{ETD}(P,Q) \leq \Delta \Rightarrow \delta_f(P,Q) \leq \Delta$ but not vice versa.

This is useful because we can insert it in our simplification algorithm to greatly speed up the Fréchet decision it has to make, furthermore, we can remember the Fréchet distance it computes and reuse it to speed up the Fréchet decision problem itself.
3.3. Valley searching

A different type of greedy algorithm is the valley searching algorithm. This algorithm is slightly more intelligent than ETD. Much like ETD, it tries to construct a single path through the freespace to try and traverse the freespace diagram, establishing its efficient linear runtime, but unlike ETD this path is not simply the diagonal. For Valley Searching, the greedy choice consists of choosing the lowest Fréchet distance among the 3 neighbours it can choose to traverse given a certain cell. As such, the algorithm will follow the path of lowest Fréchet distance, a valley. Valley searching is not implemented for this thesis, but is a very promising direction for future work.

3.3.3 Greedy experiments

Pruning percentage on querysets

We can measure the pruning percentage of the greedy algorithm similarly to the simplification experiments earlier. It should be noted that the greedy algorithm, being an upperbound to the Fréchet distance, can never discard a trajectory comparison: it can only produce a success result or an undecided result. Because of this, the pruning percentage is the percentage of successful comparisons, the comparisons where \( \text{ETD}(P, Q) \leq \Delta_{\text{query}} \). For these experiments, the ETD algorithm is run on the query sets, measuring for each set the number of comparisons the ETD algorithm can decide, divided by the total number of interesting comparisons. An interesting comparison is a comparison where the query delta is greater than the largest endpoint distance.

About 70% on sample
About 0.25% on tdrive
About 0.5% on wchars

These results may seem low, but taking into account that the greedy algorithm can only answer true, it is actually fairly high. Interestingly, the sample queryset shows us that if a trajectory comparison is easy, it can often be solved using a greedy algorithm.

Difference between greedy calc and actual Fréchet distance

We can also measure the error rate of ETD distance compared to the Fréchet distance itself. This error is potentially unbounded, however the plots in Figure 3.14 show show very clearly that the overwhelming majority of ETD calculations are very close to the actual Fréchet distance, showing its usefulness as a Fréchet proxy.

The experiment is run on all comparison sets, using only those comparisons which are not determined by their endpoints. The following graphs are cumulative graphs of the ratio of ETD to Fréchet distance, or as a formula: \( \frac{\text{ETD}(P, Q)}{\delta_f(P, Q)} \).
CHAPTER 3. PRUNING METHODS

Figure 3.14: Cumulative plot of the ETD error in Fréchet distance. For example in the sample comparison set, in nearly all comparisons the ETD is not more than 30% greater than the Fréchet distance. This error can become quite large, especially for the more complicated trajectories in the tdrive set. However, for a reduction of quadratic to linear computation complexity, this error is still very usable.
Chapter 4

Fréchet optimizations

As described in the overview section, the Fréchet distance can be solved by solving the freespace diagram with the column based algorithm from Alt and Godau [3]. Unfortunately, this solution takes quadratic time and therefore scales very poorly for larger trajectories. Where previous chapters tried to altogether avoid this computation by using pruning, this chapter aims to improve the efficiency of the computation itself. The chapter is divided in two major sections: a description of queue algorithms, which seek to reduce the part of the freespace that is calculated by finding which space is useful to compute, and freespace jumps, which can be used to skip calculations within the reachable space itself.

4.1 Queue algorithm

The freespace diagram, which can be used for solving the Fréchet decision problem, can be solved in practice using column based algorithms, but these algorithms have two major downsides. First, the column based algorithms need enough memory to store the smallest of the two trajectories, since they must store the full column in memory. Second, the column based algorithms are inefficient due to their quadratic runtime complexity. This section describes how the column algorithms can be improved to be both faster and require less memory.

4.1.1 Column based Fréchet algorithms

To solve the freespace diagram, we need to find a path in the freespace from the bottom left to the upper right corner. An obvious solution to the problem would require us to calculate the entire freespace diagram, and then attempt to find a path through the freespace. This requires quadratic memory as well as time, which are problems that column algorithms try to solve.

A column based solution to the freespace diagram computes the freespace diagram one column at the time, from bottom to top, using the information from the previous column. These algorithms depend on the observation that the only information needed to calculate the reachability of a cell in the freespace diagram are the cells directly below, to the left, and to the bottom left of the target cells. In this way, the required memory reduces to be the smaller of the two trajectories, so it has linear space complexity. Naturally, since all columns of the freespace diagram still need to be computed, left to right, the quadratic time complexity remains, and actually likely holds for any decision Fréchet algorithm [6].
4.1.2 Calculating reachability

A column based algorithm can be further improved using a few trivial observations, to arrive at the implementation used in this thesis, which we call a Queue algorithm. The Queue algorithm reduces memory requirements and increases efficiency further by calculating only the reachable space, and storing the reachable cells in a memory efficient queue. It is obvious that only the reachable space of the freespace diagram needs to be computed, as the blocked space can never support a path through the freespace diagram it is useless to retain or obtain information about it. In effect, this reduces the number of cells that the algorithm has to consider, which in turn reduces the computation time needed to compute the freespace diagram.

Moreover, especially those freespace diagrams which are hard to compute using approximation algorithms and so will survive the pruning steps, namely those where the number of free cells is small, will be quickly computed using a queue based approach with small memory requirements, whereas simpler free spaces with many free cells can be expected to be solved by other pruning methods.

4.1.3 Space efficiency

The next step in this improvement is to reduce the memory needed to store this improved column datastructure. Instead of storing all reachable cells, which may be much smaller than the size of the column but can still be a significant amount, we can choose to store runs of reachable space. That is, vertical strips of cells for which the left edge in the freespace diagram cell is completely reachable. A strip of \( n \) cells would then have \( n \) of these reachable edges in a row, and so the entire left border along these \( n \) cells would be reachable. We can compress these strips is because all the cells in a strip are identical (from the viewpoint of the next column), so we can indicate them simply by a starting cell and a run-length. This adaptation reduces the memory used by the algorithm dramatically, although it does not reduce the computation time itself, since all cells in a reachable strip still need to be iterated over, and the the availability of a reachable strip in the previous column does not directly allow us to skip cell computations, although a solution for this is discussed in the next section.

4.1.4 Queue experiments

In the following experiments the efficiency of the queued algorithm is measured against the efficiency of a normal column algorithm. The problem in these experiments is for both algorithms to calculate the freespace diagram for two trajectories \( P \) and \( Q \) with \( \Delta = r * \delta_f(P, Q) \) with \( r \) varied over the horizontal axis. The comparisons are sourced from the comparison datasets, and we pose the following restrictions for counting test cases:

- The Fréchet decision is non-trivial: is not decided based on endpoints. This property makes the result more interesting, as trivial comparisons would evaluate the same in both algorithms.

- The query delta varies around the Fréchet distance: this requirement ensures an interesting experiment, as it covers both the cases where the queued algorithm is obviously better: when the reachable space is small, and allows us to extend the query delta to the point where the queued algorithm is no longer significantly better.
It should be noted that in these tests, the column algorithm is assumed to calculate complete columns until it encounters no more reachable space. In other words, the column algorithm is not blindly calculating the entire freespace diagram, but intelligently deciding when more calculations are unnecessary. This is done to make it a more realistic and fair comparison. Note that for the memory experiments, the queued algorithm does not contain the added benefit of reachable strips, the effect of this improvement is examined in the next section.

It can be observed from both Figure 4.1 and 4.2 that the queued algorithm consistently uses less memory and calculates less rows than a generic column algorithm, but the effectiveness depends greatly on the complexity of the trajectories and the used ratio $r$.

Figure 4.1: Plotting ratio $r$ horizontally vs the percentage of the computed cells of the queued algorithm compared to the column based algorithms. The number of computed cells is measured exactly for the queued algorithm, and the column algorithm computes it as all cells for each column until there is no more reachable space. The sample comparison set graph starts at 49% because no interesting comparisons could be found for a lower value. All graphs have a slight jump for $r = 1$, which is when the query delta equals the Fréchet distance.
CHAPTER 4. FRÉCHET OPTIMIZATIONS

Figure 4.2: Plotting ratio \( r \) horizontally vs the percentage of the memory needed of the queued algorithm compared to the column based algorithms. For the queued algorithm, the required memory is the largest of the column size when computed the freespace diagram, for the column algorithm it is simply the number of vertices for the first trajectory in the comparison. The sample comparison set graph starts at 49% because no interesting comparisons could be found for a lower value. All graphs have a slight jump for \( r = 1 \), which is when the query delta equals the Fréchet distance.

4.2 Freespace jumps

The queue based algorithm solves the problem of high memory requirements, and is also fairly efficient at reducing the computation time by limiting exactly what cells of the freespace diagram are useful to compute. As described, the number of freespace cells might be expected to be low when two trajectories are similar, but exceptions to this happen often in practice, where a bottleneck in the freespace diagram makes it impossible for error-based pruning strategies to discard a trajectory pair that has a large amount of free and reachable cells in its freespace.

These cases can be improved with the observation that freespace is often predictable: if a row of freespace cells is present in a column, the same cells may likely be free in the next column. We
4.2. FREESPAC E JUMPS

can use this information, along with the triangle inequality, to skip sections of free-space column without computing all intermediate cells. We call such a skip a freespace jump, see Figure 4.3. Using freespace jumps is useful because if we would be able to reduce the time spent per column to constant time in practice, we would reduce the Fréchet decision problem to linear time in practice. Of course, there is a cost and precondition associated with a freespace jump, and these computations only makes sense if it can happen in constant time, as a linear time requirement would be comparable to simply computing the reachable cells we wish to skip in the first place, anyway. The following sections will explain how freespace jumps work, why they can be calculated efficiently, and what they mean for the freespace diagram.

Figure 4.3: An example of a freespace jump, in the freespace diagram. In blue, the reachable free cell edges. In bold black lines, the non-free edges, in normal lines, the free edges. The arrow represents a possible jump. The jump is allowed here, because it does not overshoot the reachable strip in the column before it, the only other condition is that the triangle inequality holds on the subtrajectory corresponding to the skipped cells, and the vertex corresponding to the column.

4.2.1 Computing freespace jumps

The freespace jump relies on the triangle inequality between a point and a segment, and a segment and the original trajectory. In terms of the original trajectories $P$ and $Q$, a freespace jump along a column is the traversal of one subtrajectory of $P$ while staying at the same vertex for $Q$ (otherwise we would be jumping horizontally as well). To see if such an operation is permitted for query
CHAPTER 4. FRÉCHET OPTIMIZATIONS

delta $\Delta$, subtrajectory $P_{sub}$, and vertex $v_Q$, we can obtain the equation:
\[
\delta_f(v_Q, P_{sub}) \leq \Delta
\]

When this equation evaluates to true, the freespace jump is allowed, otherwise it is intersected
by non-free space. This equation is unfortunately of linear time complexity, because of the Fréchet
distance computation of the subtrajectory. This, in turn, makes it useless, as the goal is to be more
efficient than linear time.

Fortunately, the greedy simplification algorithm used to preprocess the trajectory database has
already computed a large set of Fréchet distances to many different $P_{sub}$. To understand why, we
have to consider the greedy choice made by the simplification algorithm of Agarwal et al. [2],
which computes the Fréchet decision problem between an edge $e$ of the simplification and an ever
growing subtrajectory of the original trajectory, until the decision problem prohibits the algorithm
from continuing, and the process restarts. Combined with our choice of Fréchet proxy, which
computes an upperbound to the Fréchet distance and not simply the decision problem, we have
already computed several tuples $(e, P_{sub}, \Delta_e)$ of a (potential) simplification edge $e$, a subtrajectory
$P_{sub}$ and a Fréchet distance upperbound $\Delta_e = \text{ETD}(P_{sub}, v_Q) \geq \delta_f(P_{sub}, v_Q)$. If we now remem-
ber these tuples from the simplification stage, using that ETD is an upperbound to the Fréchet
distance, we can use these tuples along with the triangle inequality to rework the equation to the
following:
\[
\delta_f(v_Q, P_{sub}) \leq \delta_f(v_Q, e) + \Delta_e \leq \Delta
\]

for each tuple: $(e, P_{sub}, \Delta_e)$

computed by the greedy simplification.

This equation is solvable in constant time, because $\delta_f(v_Q, e)$ is a constant time operation (the
maximum of two Euclidean distances), and $\Delta_e$ is now a lookup of a value that has been computed
earlier. In conclusion, the entire inequality can now be computed in constant time, but the use of a
Fréchet upperbound has introduced an error, and the equation works only for specific tuples, and
so only for specific subtrajectories.

However, it should be noted that there are many such tuples in practice, much more than there
are edges in the simplification, as any greedy choice discarded by the simplification process still
produces a valid jump tuple. It should also be noted that these tuples vary greatly in jump length,
error and starting point, since they can be combined from all different simplification levels and
thresholds, and all different stages of the simplification algorithm. This makes them work well in
practice, as can be seen in the experiment section.

Unfortunately, this equation cannot be applied directly to execute a jump in the freespace dia-
gram and come to an exact Fréchet decision. To understand why, imagine that after taking a
freespace jump in column $c$, we end up in vertex $v$. If the cell above $v$ in $c$ has a partially free
border called $b$ in $c$, not touching vertex $v$, and the previous column $c - 1$ does not have a free
dge at $v$, we cannot count the border $b$ as reachable, but this is an error. When taking the jump,
the bottom edges of the freespace cells we skipped may have propagated reachability information
which allows us to count $b$ as reachable, but this information is now lost, thus by taking the jump
we no longer have an accurate solution.

This final problem is solved inherently by the datastructure used for the reachable strips. We
can restrict our jumps to only occur inside the reachable strip of the previous column. That is,
if the previous column $c - 1$ has a reachable strip from $i$ to $j$ we may only jump in column $c$
between $i$ and $j$ and never outside of it. This holds for each reachable strip. To see why this
works, consider two columns, the first a reachable strip, the second a jump inside that strip. The
edge case discussed leading to the loss of information is now impossible, since the reachable strip

Fréchet computations
allows us to reconstruct all reachability information, as it includes the cell vertices it crosses, and so no information can be lost.

### 4.2.2 Jump graph

We can do more with the freespace jumps than just help traversing the freespace columns quicker. To follow this reasoning, we must think about a freespace jump as line segment, with start and endpoint and a certain cost to take the jump. In the previous cases, the jump cost was binary: we can either take the jump or we cannot, but this is not necessarily the case. We can also compute the Fréchet distance (or an upperbound to it) needed to be allowed to take the jump, clamped to be always positive. Regardless of how we use the edges, if we now choose to connect all freespace jumps with shared vertices, we obtain a *jump graph*. This jump graph is a directed acyclic graph whose traversal represents a sequence of freespace jumps. This sequence of freespace jumps then corresponds to a particular path through the freespace diagram, which means that finding such a graph traversal is equivalent to solving the freespace diagram, in the decision case, or potentially computing the Fréchet distance.

Because the jump graph is a regular graph, and because we can highlight a source \((0,0)\) and target \((p,q)\) node, the Fréchet decision problem can in this way be reduced to a pathfinding problem. Unfortunately, normal deterministic shortest path finding approaches like \(A^*\) are not very effective, because the graph is too large to explore completely, and because we are not looking for the path with lowest summed weight, but with lowest maximum weight (or in case of the decision problem, 0 sum). Instead a heuristical greedy approach seems promising, which tries to find a path but gives up if it is impossible.

The advantage of such a graph algorithm is that, because it explores a jump graph created by all simplifications, it considers all simplifications at once when it is being executed. Two major downsides of this graph are that: 1, it is a discrete algorithm. Freespace jumps only work from vertex to vertex on the corner of a freespace cell, and so the resulting Fréchet decision (or distance) is not the true Fréchet decision (or distance). 2, it cannot produce a false result unless the entire graph is explored which, as mentioned previously, is prohibitively large. The first problem cannot be solved, but the second problem can be partially mitigated by traversing the blocked-space graph. The blocked space graph is a datastructure analagous to the jump graph but traversing the blocked space. Traversing the blocked space is more complicated than traversing the freespace, which is also complicated.

Unfortunately this thesis does not experimentally investigate the effectiveness of jump graphs, nor does it contain a working algorithm to solve them efficiently. Future work should investigate if the jump graph is worth exploring.

### 4.2.3 Freespace jump experiments

The following experiments analyse the improvements over the queued algorithm when implementing freespace jumps. The experiments cover all datasets, and treat both the reduction in computational complexity as well as memory reduction. All experiments are evaluated on the comparison sets, solving the decision problem, and using \(\Delta_{\text{query}} = 2 \delta_f(P, Q)\) for comparison \(P, Q\). The factor of 2 is introduced to give the resulting freespace diagram enough reachable freespace for the jumps to be useful.

The first set of graphs in Figure 4.4 show histograms of the factor of reduction in calculated cells for the freespace diagram. For instance, if for a particular comparison the jump algorithm needs to
calculate only 1 cell for every five cells in the queued algorithm, the comparison would land in the .2 bar.

These graphs show that the number of cells calculated is reduced significantly by the freespace jumps, but not very predictably, with edgecases where comparisons both benefit greatly and not at all from the addition of freespace jumps. Note that this figure is only an indication of computational improvement, as it shows a metric which is correlated to, but not the same as running time. Most notably this metric omits the additional cost of looking for viable jumps and executing them. For these reasons, a sample of 0.2 does not mean that the comparison evaluated 500% faster on the jump algorithm.

![Graphs showing reduction in computed cells](image)

**Figure 4.4**: A histogram of the reduction in computed cells when using jumps, for all comparison sets. Across all sets, there seems to be a five-fold reduction in the number of cells computed, compared to using the queued algorithm.

As mentioned, the jump algorithm also has an additional memory optimization compared to the regular queued algorithm. By only storing strips of reachable cells, instead of individual cells, the jump algorithm can significantly reduce its memory usage in practice. This memory usage improvement can make it more usable for larger trajectories, or in environments where memory
is limited or costly to access, such as GPUs. For these experiments, the memory is calculated as
the maximum number of strips in the queue for the jump algorithm, and the maximum number
of cells in the queue for the queued algorithm. Keep in mind that this improvement is required
for the jumping algorithm, but can also be applied to the queued algorithm without introducing
jumps. As such, this improvement should not be seen as an improvement of the jumps, but rather
the result of applying the memory reduction technique.

Figure 4.5 shows cumulative graphs of the factor of reduced memory when comparing the
queued algorithm to the jumping algorithm. The memory optimization technique is apparently
very effective for the used query deltas. This is expected, since the query deltas were chosen
specifically such that they would create freespace diagrams with much reachable space, which
can then be compressed using the strips. Overall the memory usage is reduced to a fraction of the
original.

![Figure 4.5: A cumulative graph of the reduction in memory by using freespace jumps. For the Sample comparison set, 97% of the samples can be computed using just 3% of the memory compared to the queued algorithm, which in turn uses less memory than the column algorithm. Overall, it is clear that using the reachable strips is very space effective in this experiment.](image)
Chapter 5
Algorithmic composition

In the previous sections, we have explored many approaches to approximating the Fréchet problem, as well as a few ways to optimize exact Fréchet calculations. In this section, we will combine these approaches into usable Fréchet algorithms which can efficiently calculate either the Fréchet decision problem or the Fréchet distance exactly. This can be done by chaining together various pruning steps, in order of difficulty of computation, and finally, if all pruning steps fail, using a column based freespace solution to the Fréchet decision problem, augmented with the optimizations from previous sections. The goal of this section is to provide evidence that the discussed methods of optimization are effective on the problem sets discussed in the overview section, and thereby also effective in practical Fréchet problems.

5.1 Efficiency measurements

To test which combination of algorithmic steps is optimal for the constructed query sets, we can test the runtime performance of all algorithms on the same problem sets. The algorithm with least pruning time should be considered the best algorithm, since taking least total time would include the preprocessing time, which makes the result dependant on the query set size. However, there is little difference in practice for these datasets, since the preprocessing is short in comparison to the pruning time. Figure 5.1, 5.2 and 5.3 show the total algorithm run times of all tested algorithms on the different problem sets, with varying query sizes.
CHAPTER 5. ALGORITHMIC COMPOSITION

The pruning steps are named as follows:

- **DF** the decision Fréchet computation
- **DiHash** pruning step using regular grids for endpoint-derived lowerbounds
- **BBOX** pruning using bounding box derived lowerbounds, no datastructure
- **Greedy** pruning step using Equal Time Distance
- **GreedyDecision** pruning step using Equal Time Distance, with early bail based on query \( \Delta \)
- **Simplification** \( i \) pruning step using sequential simplifications, with \( i \) levels

The algorithms are composed as follows:

- **BasicAlgorithm** \( DF \)
- **DiHashAlgorithm** \( DiHash - DF \)
- **DiHashGreedyAlgorithm** \( DiHash - Greedy - DF \)
- **DiHashSimplificationAlgorithm** \( DiHash - Simplification - DF \)
- **DiHashSimplificationGreedyAlgorithm** \( DiHash - Simplification - Greedy - DF \)
- **DiHashGreedySimplificationAlgorithm** \( DiHash - Greedy - Simplification - DF \)
- **DiHashBBOXSimplificationGreedyAlgorithm** \( DiHash - BBOX - Simplification - Greedy - DF \)
- **DiHashBBOXSimplificationGreedyDecisionAlgorithm** \( DiHash - BBOX - Simplification - GreedyDecision - DF \)
Figure 5.1: Logarithmic total running time plots for all algorithms on the sample queryset, the right side of the figure shows the different algorithms sorted by efficiency from top to bottom, showing an overall 50 times improvement between naive approaches and the best algorithm.
Figure 5.2: Logarithmic total running time plots for all algorithms on the tdrive queryset, the right side of the figure shows the different algorithms sorted by efficiency from top to bottom, showing an overall 30 times improvement between naive approaches and the best algorithm.
5.1. EFFICIENCY MEASUREMENTS

Figure 5.3: Logarithmic total running time plots for all algorithms on the wchars queryset, the right side of the figure shows the different algorithms sorted by efficiency from top to bottom, showing an overall 6 times improvement between the already fast naive approaches and the best algorithm.
CHAPTER 5. ALGORITHMIC COMPOSITION

5.2 Analysis

The previous graphs show us exactly how the various pruning steps behave on the different datasets when composed inside complete algorithmic solutions to the Fréchet query problem. These graphs show some interesting results, which are discussed in this section.

First of all, the spatial hash only appears to help the wchars dataset much 5.3 (see difference between BasicAlgorithm and DiHashAlgorithm), and offers no runtime improvements in the other sets. This could be explained by the relatively small dataset size: with only 10000 trajectories, looping over each of them and performing a constant time operation is not that much work compared to accessing the spatial hash datastructure. Also, the query deltas in the querysets may influence the behavior by intersecting with too many grid cells.

Also, the greedy algorithm seems to work fairly poorly as a pruning solution for the more complex trajectories. From the experiments it was clear that it is not the best approximation of the Fréchet distance, and the performance can be expected to be better for shortest path trajectories, as the diagonal in the freespace is more likely to be open. Luckily this is also the case, in Figure 5.1 (DiHashGreedyAlgorithm).

Another simple observation is the impact of simplification. Just a single simplification level helps tremendously across all datasets, with multiple levels being helpful in general, up to level 3. It should be noted that these algorithms are ran with the simplification ratios learned during the experiments. For a fairer comparison, future work could attempt to test a single set of ratios for any dataset. That being said, the learned ratios are not very different from one set to another, which can reduce the unfair advantage of these experiments.

Then, the Bounding Box pruning apparently helps all datasets somewhat, which can be expected as a constant time lowerbound pruning metric, but its performance on the tdrive dataset is very promising. This can be explain by the complexity of the trajectories, which may result in very different boundingboxes compared to the shortest-path properties in the sample set, or the limited dimensions of the trajectory database in the wchars dataset.

Finally, an important observation is that the experiments show a clear optimal algorithm for all querysets: DiHashBBOXSimplificationGreedyAlgorithm (or DiHashBBOXSimplificationGreedyDecisionAlgorithm, it appears to be a tie). This is important because it shows that the goal of this section has been achieved: the pruning and optimization methods mentioned in this thesis can, when composed properly into an algorithm, solve the practical problems generated and discussed in the overview section in an efficient manner. Moreover, the best algorithm is composed of all pruning steps in some form or another, which may indicate that the algorithm could be extended with additional pruning steps, or may indicate that the algorithm is sufficiently general purpose to be applied somewhat efficiently to any dataset.

Of course, these conclusions cannot be drawn yet, and would require more research into the optimization strategies proposed, and the Fréchet query problem in general.
Chapter 6

Conclusions

In this thesis, we have looked at accurately solving the following problem: given a database of trajectories $\mathcal{P}$, and a number of query pairs $(Q, \Delta)$ with $Q$ a query trajectory and $\Delta$ a distance, report all trajectories $P \in \mathcal{P}$ where $\delta_f(P, Q) \leq \Delta$. To solve this problem, we have shown how to generate practical datasets for the Fréchet problem, how to employ approximation algorithms to prune the input dataset, and how to further optimize the ubiquitous column algorithms used to solve the freespace diagram, and with it the Fréchet decision problem. We have shown that a limited set of pruning steps, motivated by spatial hashes, intelligent simplifications and greedy approaches, can reduce huge sets of comparisons to a fraction of their original size, retaining only those simplifications which are inherently difficult to solve. Improvements over previous column based approaches have shown that even these few remaining difficult trajectory comparisons can be solved efficiently. The use of freespace jumps, a novel approach to optimizing the decision Fréchet problem, is particularly effective.

This thesis has not only yielded an analysis of strategies for solving the Fréchet problem, but also a complete implemented, tested and optimized algorithm that reached second place in the ACM sigspatial GIScup of 2017 [9]. This resulting Fréchet query algorithm can solve millions of realistic trajectory comparisons very quickly, taking between 10 seconds to a few minutes on a modern system depending on trajectory complexity, and as such the algorithm itself or any of its components can serve as a basis for solving Fréchet problems efficiently in practice. Most importantly, the methods developed in this thesis and by other teams for the ACM sigspatial GIScup competition should in the future be united into a canonical solution for Fréchet query algorithms.

Future work should aim to further explore the relationship between simplification ratio and suitability for Fréchet decision problems, construct more flexible and performant greedy approaches, and analyse the usefulness of the jump graph as a substitute to the freespace diagram.
Bibliography


Fréchet computations


[16] M. Lichman. UCI machine learning repository, 2013. 6


Fréchet computations